第 5 章

CHAPTER 5

程 序 结 构

C++的程序由一个或多个函数组成，要对 C++的程序结构有一个全面的了解，就必须掌握如何将多个函数组成模块，几个模块如何构成程序，以及如何在模块之间共享数据和调用同一工程文件中其他模块的函数的方法。掌握这些方法的基础是理解程序编译的过程以及系统内存空间的分配和释放。

学习目标

- 区分各种变量的类型及其分配的内存空间；
- 掌握全局变量、一般局部变量的特点和使用；
- 掌握静态局部变量的特性；
- 能够区分并综合运用不同类型的变量；
- 理解作用域、可见度和生存期的概念；
- 掌握不同作用域的范围；
- 掌握几种预处理命令的用法。

5.1 全局变量与局部变量

C++程序中的变量根据定义位置的不同，其可见度和生存期也不同。可见是指变量可以被正常使用，可见区域也称为作用域；生存期是指变量占用的内存单元。某些变量可能在生存期内某时刻占用内存单元，却不能被使用(即不可见)。根据定义位置的不同，变量可以分为全局变量和局部变量。

5.1.1 内存区域的布局

C++程序运行时所占用的内存空间分为 4 个区域，程序运行时内存空间的分配如图 5-1 所示。

图 5-1 程序运行时内存空间的分配图

(1) 代码区，存储程序的可执行代码，即程序中的各函数代码块。

(2) 全局数据区，存储一般的全局变量和静态变量(静态全局变量、静态局部变量)。该区的变量在没有初始化的情况下被自动默认为 0。

(3) 栈区，存储程序的局部变量，包括函数的形参和定义在函数内的一般变量。分配栈

区时，不处理原内存中的值，即栈区中的变量在没有初始化的情况下，其初值是不确定的。

(4) 堆区，存储程序的动态数据，堆区中的数据多与指针有关。分配堆区时，也不处理原内存中的值。

5.1.2 全局变量

全局变量是指定义在所有函数体外的变量，在整个程序中都是可见的，能够被所有函数所共享。全局变量存储在全局数据区，在主函数 main()运行之前就已经存在了，如果其在定义时没有给出初始值，则自动初始化为 0。例如：

```
int i = 10;                                          //全局变量
void sub()
{
    i = i + 10;
    cout << i;
}
```

【例 5-1】 全局变量应用举例。

```
1   /*****************************************************
2                程序文件名:Ex5_1.cpp
3                全局变量应用举例
4    *****************************************************/
5   #include <iostream>
6   using namespace std;
7   int i = 10;                                      //全局变量
8   void add()
9   {
10     i = i + 10;
11     cout <<" i = "<< i << endl;
12  }
13  main()
14  {
15    i = i + 5;
16    cout <<" i = "<< i << endl;
17    add();
18    i = i + 6;
19    cout <<" i = "<< i << endl;
20  }
```

```
i= 15
i= 25
i= 31
```

图 5-2 例 5-1 运行结果

程序运行结果如图 5-2 所示。

【程序解释】

(1) 第 7 行定义了全局变量 i，定义在所有函数之外。

(2) 程序从入口第 13 行 main()函数开始，第 15 行执行了 i=i+5 后，全局变量 i 的值变为 15。

(3) 程序执行到第 17 行调用 add()函数后，跳转到 add()函数定义体执行 i=i+10，此时全局变量 i 的值变为 15+10=25；因为全局变量对程序中所有函数是可见的，所以如果程序中的某个函数修改了全局变量，则其他函数仅能“可见”并共享修改后的结果。

（4）执行完 add()函数后返回到函数调用点执行其后续语句，即 i=i+6，这时，全局变量 i 的值变为 25+6=31。

全局变量通常定义在程序顶部，其一旦被定义后就在程序中的任何位置可见，C++允许在程序中的任何位置定义全局变量，但要在所有函数之外。全局变量对其定义之前的所有函数定义是不可见的。例如，若例 5-1 中第 7 行全局变量的定义位置修改到 add()与 main()之间，即：

```
void add()
{
  i = i + 10;
  cout <<" i = "<< i << endl;
}
int i = 10;                                          //定义位置修改到此处
main()
…
```

这时，程序就会提示 add()函数中“i 未定义”的编译错误。

5.1.3 局部变量

局部变量是指在一个函数或复合语句中定义的变量。局部变量在栈中分配空间，其类型修饰是 auto，但习惯上通常省略 auto。局部变量仅在定义它的函数内，从定义它的语句开始到该函数结束的区域是可见的。

由于函数中的局部变量存放在栈区，在函数开始运行时，局部变量在栈区被分配空间，函数结束时，局部变量随之消失，其生命期也结束。在一个函数内可以为局部变量定义合法的任意名字，而无须担心与其他函数中的变量或者全局变量同名。当某函数中的局部变量与全局变量同名时，在该函数内部局部变量的可见区域中，局部变量的优先级要高于同名的全局变量，即局部变量可见，而同名的全局变量不可见。

如果局部变量没有被显式初始化，其值是不可预知的。

【例 5-2】 局部变量应用举例。

```
1   /**************************************************
2                  程序文件名:Ex5_2.cpp
3                  局部变量应用举例
4    **************************************************/
5   #include <iostream>
6   using namespace std;
7   int i = 10;                                        //全局变量 i
8   void add()
9   {
10     i = i + 7;                                      //全局变量 i
11     cout <<" i = "<< i << endl;                     //全局变量 i
12     int i = 3;                                      //局部变量 i
13     i = i + 10;                                     //局部变量 i
14     cout <<" i = "<< i << endl;                     //局部变量 i
15  }
16  main()
```

```
17  {
18      i = i + 5;                                  //全局变量 i
19      cout <<" i = "<< i << endl;                 //全局变量 i
20      add();
21      i = i + 6;                                  //全局变量 i
22      cout <<" i = "<< i << endl;                 //全局变量 i
23  }
```

程序运行结果如图 5-3 所示。

```
i= 15
i= 22
i= 13
i= 28
```

图 5-3 例 5-2 运行结果

【程序解释】

(1) 第 7 行定义了全局变量 i,定义在所有函数之外。

(2) 在 add()函数中,第 12 行定义了同名的局部变量 i,并初始化为 3。局部变量 i 仅在 add()函数中是可见的,其生命期从第 12 行开始,到第 15 行结束。根据同名变量的特点可以知道,在此区域内同名的全局变量 i 不可见。

(3) 在 add()函数中,局部变量 i 定义之前的变量 i 系统都认为是全局变量,如第 10 行和第 11 行的语句是对全局变量 i 的操作。

(4) 局部变量与其同名的全局变量是相互独立的变量,局部变量存储在栈区,而全局变量存储在全局数据区。

(5) 第 18 行全局变量 i 的值为 10+5=15;之后调用 add()函数,第 10 行全局变量 i 的值为 15+7=22,第 13 行局部变量 i 的值为 3+10=13;返回到函数调用点执行第 21 行全局变量 i 的值为 22+6=28。

当定义局部变量的函数又调用了其他函数,例如:

```
…
1   int i;
2   void sub()
3   {
4     i = i - 5;
5   }
6   void add()
7   {
8     int i = 3;
9     i = i + 6;
10    sub();
11  }
12  main()
13  {
14   …
15   add();
      …
```

上面程序段中定义了全局变量 i,在 add()函数中定义了局部变量 i;main()函数调用 add()函数,add()函数又调用 sub()函数;局部变量 i 的可见区域仅为第 8～11 行,而被调用的 sub()函数中的代码不是局部变量 i 的可见区域,所以第 4 行是对全局变量 i 的操作。

局部变量包括函数的形参、函数内定义的变量和复合语句内定义的变量;由于局部变量具有一定的范围局限,所以在不同的函数中可以定义同名的变量,这些变量之间没有任何

逻辑关系，不会相互影响。


视频

5.1.4 静态局部变量

用 static 修饰的变量称为静态变量。定义的语法格式为：

```
static  数据类型  变量名 = 初值;
```

根据变量声明位置的不同，可分为静态全局变量和静态局部变量。静态变量存储在全局数据区，像全局变量一样，如果没有显式地给静态变量初始化，那么该变量将自动默认为 0。静态变量的初始化仅进行一次。

静态全局变量是定义在所有函数体外的静态变量，只能供本模块使用，不能被其他模块重新声明为 extern 变量(外部变量，5.2 节介绍)；静态局部变量是定义在函数或复合语句块中的静态变量。静态局部变量既具有局部变量的性质又具有全局变量的性质，即具有局部作用域和全局生命期。静态局部变量实质上是一个可供函数局部存取的全局变量。

【例 5-3】 静态局部变量应用举例。

```
1   /*************************************************
2                   程序文件名:Ex5_3.cpp
3                   静态局部变量应用举例
4    *************************************************/
5   #include <iostream>
6   using namespace std;
7   void add()
8   {
9     static int i = 0;
10    int j = 1;
11    i++;
12    j++;
13    cout<<" i = "<<i<<","<<" j = "<<j<<endl;
14  }
15  main()
16  {
17    for(int i = 0;i<5;i++)
18      add();
19  }
```

程序运行结果如图 5-4 所示。

```
i= 1, j= 2
i= 2, j= 2
i= 3, j= 2
i= 4, j= 2
i= 5, j= 2
```

图 5-4 例 5-3 运行结果

【程序解释】

(1) 第 9 行在 add()函数中定义了静态局部变量 i，并初始化为 0。

(2) 程序的执行过程为：

① 进入 main()函数，执行 for 语句的第一次循环，即调用 add()函数；

② 进入 add()函数，执行第 9 行语句“static int i=0”，定义静态局部变量 i 并赋初值；

③ 执行第 10 行语句“int j=1”，定义局部变量并赋初值，依次执行第 11～13 行的语句；

④ 结束 add()函数，返回调用点，并判断 for 循环语句是否结束，若结束跳转到第⑤步，否则，跳转到第③步；

⑤ 程序结束。

(3) 从程序的执行过程可以知道,静态局部变量的定义语句“static int i=0”在程序中仅执行一次。

(4) 因为静态局部变量的生命期与全局变量一样是全局生命期,所以每次 add()函数结束时,静态局部变量 i 的内存空间和值被保留下来,由结果可以看出,后续有关 i 的运算都是在上一次计算结果的基础上进行的;而局部变量 j 在每次 add()函数结束时都被释放,再一次调用 add()函数时,局部变量 j 被重新分配空间和初始化。

5.2　外部存储类型

一个源文件(.cpp)仅能够完整表达规模较小的程序,本节之前章节中的程序都是由单个源文件组成的。但是,实际应用中的程序大多由若干个源文件组成,几个源文件分别编译成目标文件(.obj),最后连接成一个完整的可执行文件(.exe)。这些源文件一般添加在同一个工程文件(.prj)中,C++规定,同一工程的多个文件中,有且只有一个源文件包含主函数 main(),否则会出现程序入口的二义性。

如果存在同一工程的多个文件共同使用的全局变量,就在其中的一个文件中定义全局变量或函数,在其他文件中使用“extern”关键字进行声明,其作用既通知了编译器该变量或函数已经被定义过,又避免了重复定义的错误发生。被使用“extern”说明的变量或函数是外部文件。语法格式如下:

```
extern 数据类型变量名\函数原型;
```

如图 5-5 所示为工程文件的创建图。工程文件的创建方法是:

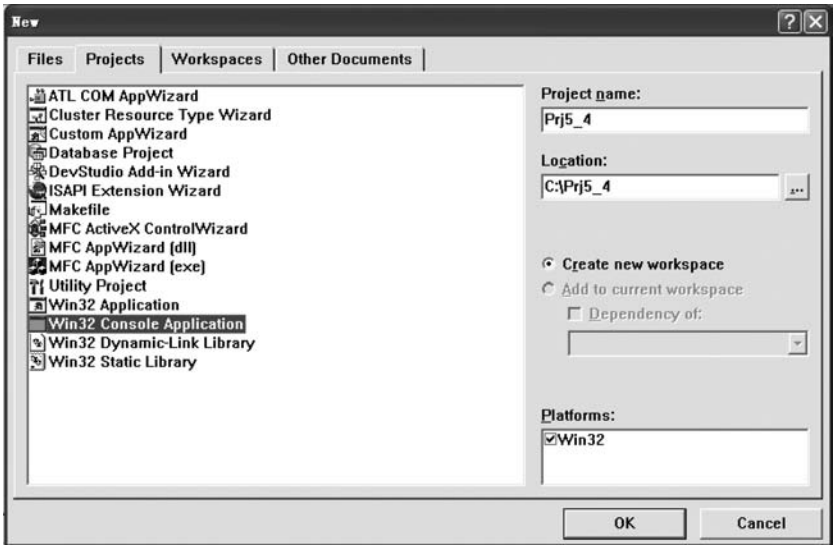

图 5-5　工程文件的创建

(1) 选中 Projects 标签的“Win32 Console Application”(Win32 控制台应用)项;

(2) 选择存储工程文件的磁盘位置(Location)并给工程命名(Project name),工程名和

其包含的文件名可以同名。

【例 5-4】 外部存储结构应用举例。

```
/*****************************************************
                工程文件名:E5_4.prj
                源文件名:Ex5_4_1.cpp
                      Ex5_4_2.cpp
                外部存储结构应用举例
 *****************************************************/
//Ex5_4_1.cpp
#include <iostream>
using namespace std;
void fun1();
extern void fun2();
int i;
main()
{
  i = 3;
  fun1();
  cout << i << endl;
}
void fun1()
{
  fun2();                                    //fun2()的定义体不在本文件中
}

//Ex5_4_2.cpp
extern int i;                                //i 定义在本工程的其他源文件中
void fun2()
{
   i = 18;
}
```

【程序解释】

(1) 在 Ex5_4_1.cpp 中定义了函数 fun1()和全局变量 i，而函数 fun2()在本工程的 Ex5_4_2.cpp 中定义，因此在 Ex5_4_1.cpp 的第 10 行声明函数 fun2()前添加了 extern，说明其为外部文件。

(2) 由于 C++默认所有函数的声明和定义都是 extern 的，所以文件 Ex5_4_1.cpp 中 extern void fun2()前的 extern 说明符可以省略，函数 fun2()的声明等价于：

```
void fun2();
```

(3) 主函数 main()仅在 Ex5_4_1.cpp 中进行了定义，由于全局变量 i 定义在 Ex5_4_1.cpp 文件中，所以当 Ex5_4_2.cpp 文件要使用全局变量 i 时，就需要在其开头声明“extern int i”，表示该变量 i 不在本文件中分配空间，而在程序其他文件中进行了定义。

(4) 在工程的各个源文件中，使用 extern 只是将其他源文件中已经定义的变量或函数声明为外部文件，可以在本文件中使用，不是对变量或函数的定义，因此不能进行赋值操作。

例如，下面写法是错误的：

```
extern int i = 8;
```

(5) 假设某个工程文件由两个或两个以上源文件组成，其中一个源文件中出现了extern修饰的变量或函数，那么在其他某个源文件中必须有对该变量或函数的定义。

5.3 作用域

作用域是标识符在程序代码中的有效范围，按其范围可分为函数原型作用域、局部作用域(块作用域)、函数作用域和文件作用域。除标号外的标识符的作用域都开始于标识符的声明处。

5.3.1 函数原型作用域

函数原型作用域是C++中范围最小的作用域。函数原型声明中的形参就在该作用域内，其范围起始于函数原型声明的左圆括号，结束于函数原型声明的右圆括号。例如，下面的函数原型声明：

```
void fun1( inti, int j = 4, int m = 6);
```

标识符i、j和m的作用域为函数原型声明内部，因此函数原型声明中形参的标识符名可以省略，保留标识符的作用在于增强程序的可读性。习惯上将函数原型声明的形参标识符与其对应的函数定义中形参的标识符保持一致。例如：

```
void fun1( inti, int j = 4, int m = 6);                    //函数原型声明
…
void fun1( inti, int j, int m)                             //函数定义
{
   …
}
```

5.3.2 局部作用域

局部作用域也称为块作用域。所谓块是指用一对花括号“{}”括起来的部分。当标识符位于某块内时，其作用域从声明处开始，到块结束处终止。

块可以进行嵌套，即块中内嵌新的块。在出现嵌套块的情况下，标识符的作用域属于包含它的最近的块。

函数定义的形参和函数体同属于一个块；语句是一个程序单位，如果语句中出现标识符的声明则标识符的作用域在语句中，如if…else语句、switch语句和循环语句等。例如，下面的程序段将出现变量重复定义的错误：

```
void func( int i)
{
  int i;                                                   //错误,变量重复定义
}
```

如果将上面程序段中的“int i”包含到新的内嵌块中，则代码是正确的，即：

```
void func( int i)
{
    {
      int i;
    }
}
```

形参中 i 的作用域从声明处开始到外层花括号结束时终止，函数体 i 的作用域从声明处开始到内层花括号结束时终止。

而语句中如果出现标识符新的定义，则标识符的作用域从新的声明处开始而不会有重复定义的错误提示。以 while 循环语句为例：

```
1   while( int i = 3)
2   {
3     int i = 6;
4     cout << i << endl;
5   }
```

第 1 行定义的 i=3 的作用域从声明处开始到第 5 行结束；第 3 行定义的 i=6 的作用域从声明处开始到第 5 行结束；当相互嵌套块中有名字相同的标识符时，则块重叠部分标识符的作用域从属于内层块，因此第 4 行的 i 是指第 3 行定义的 i，其值为 6；while()后{}中的块相当于 while 的内嵌块。

5.3.3 函数作用域

C++中，goto 语句的标号是唯一具有函数作用域的标识符。标号的声明使其在声明的函数内的任何位置都可以被使用。例如：

```
…
void fun1()
{
    goto M;
    int i = 5;
    if( i > 5)
    {
      M:
        cout <<"i = "<< i << endl;
      }
    }
…
```

5.3.4 文件作用域

既不在函数原型中，又不在块中的标识符具有文件作用域。文件作用域是在所有函数定义之外说明的，从声明处开始，到文件结束时终止。全局变量和常量具有文件作用域。

如果头文件（下节介绍）被其他源文件导入“include”，则在头文件的文件作用域中声明的标识符的作用域也扩展到该源文件，直到源文件结束。

C++中函数的定义是平行的，除了 main()函数被系统自动调用外，其他函数都可以互相调用。假设存在三个函数：f1()、f2()和 f3()，函数 f1()既可以直接调用函数 f3()，也可以间接调用它，即先调用函数 f2()，再由 f2()调用 f3()。

5.4 文件结构

5.4.1 头文件

头文件是指以.h 作为扩展名的文件，头文件里包含被同一工程文件中多个源文件引用的具有外部存储类型的声明。头文件的引入可以将工程文件中共享的变量或函数的声明集合化，每个需要使用这些共享信息的源文件只需在自身文件顶端导入头文件即可。头文件的导入使用编译预处理“#include”方法。头文件一般可以包含如下声明：

```
函数声明,如 extern int func1();
常量定义,如 const int a = 3;
数据声明,如 extern int i;
内联函数定义,如 inline int add(int i)
                {return ++i;}
包含文件,如 #include <math.h>
宏定义,如 #define PI 3.14159;
```

头文件中不能包含一般函数和数据的定义。

【例 5-5】 设计一个包含三个源文件的工程文件，源文件的功能分别为计算球体的体积、计算球体的表面积和输入球体半径并调用另外两个源文件，共享的声明存储在头文件。

```
/*************************************************
                工程文件名:Ex5_5.prj
                头文件名: myball.h
                头文件应用举例
*************************************************/
double volume(double i);
double area(double i);
const float PI = 3.14;

/*************************************************
                工程文件名:Ex5_5.prj
                源文件名: Ex5_5_1.cpp
                计算球体的体积
*************************************************/
#include "myball.h"
double volume(double i)
{
    return 4 * PI * i * i * i/3;
}

/*************************************************
                工程文件名:Ex5_5.prj
                源文件名:Ex5_5_2.cpp
```

```
                    计算球体的表面积
*************************************************/
#include "myball.h"
double area(double i)
{
    return 4 * PI * i * i;
}

/*************************************************
                    工程文件名:Ex5_5.prj
                    源文件名:Call.cpp
                    调用函数
*************************************************/
#include <iostream>
#include "myball.h"
using namespace std;
main()
{
  double radius;
  cout <<" 请输入球体的半径: ";
  cin >> radius;
  cout <<" 球体的体积为"<< volume(radius)<< endl;
  cout <<" 球体的表面积为"<< area(radius)<< endl;
}
```

```
请输入球体的半径:  4
球体的体积为267.947
球体的表面积为200.96
```

图 5-6　例 5-5 运行结果

程序执行前需要分别对三个源文件进行编译，然后与头文件进行链接，程序运行结果如图 5-6 所示。

5.4.2　编译预处理

C++中以#开头，以换行符结尾的命令行称为预处理命令。一个程序运行前，首先进行去掉注释行、变换格式等预处理操作，然后再通过编译器将高级语言编写的程序翻译成计算机能识别的机器语言。预处理命令不以分号结尾。

1. #include 命令

#include 命令也称为文件包含命令，是指在一个 C++文件中将另一个文件的内容全部包含进来。文件包含命令的语法格式为：

```
#include <包含文件名>
```

和

```
#include "包含文件名"
```

第一种形式<>表示包含的文件在编译器指定的文件包含目录中；第二种形式""表示系统首先到当前目录下查找包含文件，如果没找到，再到系统指定的文件包含目录中查找。通常情况下，<>包含的是系统提供的头文件，而""包含的是程序员自己定义的头文件。一条 include 命令只能指定一个被包含文件，如果要包含 m 个文件，就要使用 m 条 include 命令。

2. 条件编译

在有些情况下，不需要程序中的所有语句都参加编译，而是根据一定的条件去编译程序

中的不同部分，这称为条件编译。这种机制使同一程序在不同的编译条件下生成不同的目标文件。常用的条件编译指令有：

1）表达式作为编译条件

其语法格式为：

```
#if 表达式
    程序段 1
[#else
    程序段 2]
#endif
```

其中，#if 后只能是常量表达式，表示如果表达式的值不为零，则执行程序段 1，否则执行程序段 2。

2）宏名已经定义作为编译条件

其语法格式为：

```
#ifdef 宏
    程序段 1
[#else
    程序段 2]
#endif
```

表示如果宏已经被定义，则编译程序段 1，否则编译程序段 2。

3）宏名未被定义作为编译条件

其语法格式为：

```
#ifndef 宏
    程序段 1
[#else
    程序段 2]
#endif
```

表示如果宏未被定义，则编译程序段 1，否则编译程序段 2。

利用条件编译还可以在调试程序的过程中增加调试语句，借以达到程序跟踪的目的。

5.5 实例应用与剖析

【例 5-6】 全局变量、局部变量和静态局部变量的综合应用。

```
1   /**************************************************
2             工程文件名:Ex5_6.cpp
3             全局变量、局部变量和静态局部变量综合应用
4    **************************************************/
5   #include <iostream>
6   using namespace std;
7   void func();
8   int n = 1;                                    // 全局变量
9   main()
10  {
```

```
11    static int a;                                    // 静态局部变量
12    int b = -10;                                     // 局部变量
13    cout <<" a:" << a <<" b:" << b <<" n:" << n << endl;
14    b += 4;
15    func();
16    cout <<" a:" << a <<" b:" << b <<" n:" << n << endl;
17    n += 10;
18    func();
19  }
20  void func( )
21  {
22    static int a = 2;                                // 静态局部变量
23    int b = 5;                                       // 局部变量
24    a += 2;
25    n += 12;
26    b += 5;
27    cout <<" a:"<< a <<" b:"<< b <<" n:"<< n << endl;
28  }
```

```
a:0  b:-10  n:1
a:4  b:10  n:13
a:0  b:-6  n:13
a:6  b:10  n:35
```

图 5-7　例 5-6 运行结果

程序运行结果如图 5-7 所示。

【程序解释】

(1) 运行结果显示本例共有 4 行输出。

(2) 本例定义了 1 个全局变量，2 个一般局部变量和 2 个静态局部变量。局部变量互相重名，但本质上是相互独立的变量。

(3) 程序的执行过程为：进入 main()函数，在第 11 行定义了 main()中的静态局部变量 a，系统自动赋初始值为 0；第 12 行定义了一般局部变量 b，其初值为－10，因此第 1 行分别输出 main()中的 a、b 以及全局变量 n 分别为 0、－10 和 1。

(4) 第 14 行语句对 main()的一般局部变量 b 进行加 4 操作，此时 main()中 b 的值为－6；第 15 行调用 func()函数，跳转到对应的函数定义体。

(5) 在 func()函数中，定义了名为 a 的静态局部变量并赋初始值为 2，此函数中的 a 与 main()函数中的 a 分配的内存空间不同，它们之间没有任何逻辑关系。第 27 行分别输出 func()中的 a、b 以及全局变量 n，结果分别为 4、10 和 13。

(6) 执行完 func()函数后，跳回函数调用点执行第 16 行语句，此处的变量分别为 main()中的 a、b 和全局变量 n，结果分别为 0、－6 和 13。

(7) 第 17 行对全局变量 n 进行加 10 操作，即 n 的值变为 23；第 18 行第二次调用 func()函数，根据静态局部变量的特点，在执行函数体的过程中，不再执行第 22 行中变量定义语句，而直接从第 23 行开始执行；第 24 行对 a 的操作是在上次计算结果 a＝4 的基础上加 2，因此 a＝6，即输出结果第 4 行的值分别为 a＝6、b＝10、n＝35。

【例 5-7】 作用域应用举例。

```
1   /*************************************************
2           工程文件名:Ex5_7.cpp
3           作用域应用举例
4    *************************************************/
5   #include <iostream>
```

```
6   using namespace std;
7   int i = 36;                                    //全局变量
8   void func(int i = 12);
9   main()
10  {
11    cout <<" 作用域示例:"<< endl;
12    func();
13  }
14  void func(int i)
15  {
16    cout <<" 局部作用域: i = "<< i << endl;
17    {
18      int i = 13;
19      cout <<" 内嵌局部作用域: i = "<< i << endl;
20      {
21        for(int i = 14;i < 15;cout <<"for 局部作用域:i = "<< i << endl,i++)
22        {
23          cout <<" for 局部作用域: i = "<< i << endl;
24          int i = 15;
25          i++;
26          cout <<" for 的内嵌局部作用域:i = "<< i << endl;
27        }
28      }
29    }
30   cout <<" 局部作用域: i = "<< i << endl;
31  }
```

程序运行结果如图 5-8 所示。

```
作用域示例:
局部作用域:          i=12
内嵌局部作用域:      i=13
for局部作用域:       i=14
for的内嵌局部作用域: i=16
for局部作用域:       i=14
局部作用域:          i=12
```

图 5-8 例 5-7 运行结果

【程序解释】

(1) 第 8 行为函数声明,形参的定义 i=12 是范围最小的函数原型作用域。

(2) 第 16 行显示是 func()函数的局部作用域,从第 14 行开始,到第 31 行结束;第 19 行显示是 func()内的内嵌局部作用域,第 18 行定义的 i 不同于第 14 行定义的 i,作用域到第 29 行结束。

(3) 第 21 定义的 i=14 的作用域从定义点开始,到 for 语句结束时终止;但是在 for 语句内第 24 行又定义了 i=15,此处的 i 不同于第 21 行定义的 i,属于不同的块。

(4) 只有在不同的块中才能定义同名的变量,否则系统会提示重复定义的错误。

(5) 第 16、30 行的 i 对应的是第 14 行的定义,第 25 行的 i 对应的是第 24 行的定义,第 21 行的 i 对应的是第 21 行的定义,第 19 行的 i 对应的是第 18 行的定义。

5.6 建模扩展与优化

【例 5-8】 数学课上,老师让班长一诺带领大家一起复习巩固最大公约数和最小公倍数问题,一诺给全班出了一道题:请输入两个正整数 $x(2\leqslant x\leqslant 10^5)$和 $y(2\leqslant y\leqslant 10^6)$,求出满足以 x 为最大公约数和以 y 为最小公倍数的两个正整数(表示为 M 和 N)及个数。

```
1   /*************************************************/
2                   程序文件名:Ex5_8.cpp
3                   最大公约数和最小公倍数
4   /*************************************************/
5   #include<bits/stdc++.h>
6   using namespace std;
7   int gcd(int a,int b)                          //输出两个数的最大公约数
8   {
9     if(b==0)
10    {
11       return a;
12    }
13    return gcd(b,a%b);
14  }
15  main()
16  {
17      int x,y,cnt=0;
18      cout<<"请输入两个正整数 x 和 y 的值:";
19      cin>>x>>y;
20    cout<<endl;
21      for(int i=x;i<=y;i++)
22      {
23        for(int j=i;j<=y;j++)
24        {
25            int h=gcd(i,j);
26            if( h==x && i/h*j==y)
27            {
28                cout<<"满足条件的正整数 M= "<<i<<" N= "<<j<<endl;
29                cout<<"满足条件的正整数 M= "<<j<<" N= "<<i<<endl;
30             cnt++;
31            }
32        }
33      }
34      cout<<"满足条件的正整数共 "<<cnt*2<<"个"<<endl;
35  }
```

程序运行结果如图 5-9 所示。

```
请输入两个正整数x和y的值: 3 60

满足条件的正整数 M= 3 N= 60
满足条件的正整数 M= 60 N= 3
满足条件的正整数 M= 12 N= 15
满足条件的正整数 M= 15 N= 12
满足条件的正整数共 4个
```

图 5-9　例 5-8 运行结果

【程序解释】

（1）根据最大公约数和最小公倍数特点可知：x×y=M×N。

（2）第 7～14 行定义函数 gcd()来求两个参数的最大公约数。

（3）第 23 行中 for(int j=i; j<=y; j++)用简单暴力枚举法，在逻辑上可写为 for(int j=x; j<=y; j++)，此时需要删除第 29 行，同时第 34 行输出个数为 cnt 个；但这样改写后，该 for 语句的执行次数将翻倍，不满足时间复杂度最低的优化要求。

（4）第 34 行中发现了一对 M、N 实际就发现了两对 M、N，因为二者可以互换且不等，

如找到了 M=12,N=15,则同时可获取 M=15,N=12。

【例 5-9】 一诺在美术课上给马上要过生日的老师做了张贺卡,为了装饰这张贺卡,一诺买了一条彩带,但是彩带上并不是所有颜色一诺都喜欢,于是一诺决定裁剪这条彩带,以取得最好的装饰效果,请设计程序找出彩带最好装饰效果区间。

现已知彩带由 n 种不同的颜色顺次相接而成,而每种颜色的装饰效果用一个整数表示(包括正整数、0 或负整数),从左到右依次为 $a_1,a_2,\cdots,a_n$,一诺可以从中裁剪出连续的一段用来装饰贺卡,而装饰效果就是这一段上各个颜色装饰效果的总和,一诺需要选取装饰效果最好的一段颜色来制作贺卡(取该段颜色数值之和的最大值)。当然,如果所有颜色的装饰效果都只能起到负面的作用(即 $a_i<0$),一诺也可以放弃用彩带来装饰贺卡(获得的装饰效果为 0)。

```
/*************************************************/
                程序文件名:Ex5_9.cpp
                装饰彩带效果
/*************************************************/
#include<bits/stdc++.h>
using namespace std;
int n,m=0,a[1000];
main()
{
 int i,j,x,y;
   cout<<"请输入原始彩带上的颜色条数:";
   cin>>n;
   cout<<"请输入"<<n<<"条颜色的数值表示:"<<endl;
   for(i=1;i<=n;i++)
   {
     cin>>a[i];
   }
   for(i=1;i<=n;++i)
   {
     int tmp =a[i];
     for(j=i+1;j<=n;++j)
   {
   tmp+=a[j];
   if(tmp>m)
   {
     m=tmp;
     x=i;
     y=j;
   }
     }
   }
 cout<<"获得最佳彩带装饰效果区间为第"<<x<<"段到第"<<y<<"段"<<endl;
   cout<<"最佳彩带装饰效果值为"<<m<<endl;
}
```

程序运行结果如图 5-10 所示。

【程序解释】

(1) 第 12 行输入彩带的颜色条数,如彩带颜色顺次为红、黄、蓝、红、绿、黄,则条数为 6,不同颜色的数值不同。

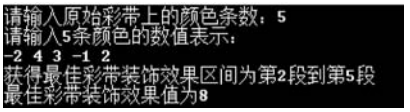

图 5-10　例 5-9 运行结果图

（2）第 14～17 行输入不同彩带颜色的数值。

（3）第 27 行和第 28 行使用变量 x 和 y 存储当前彩带颜色数值累加和最大区间的起始点和终点的位置。

（4）第 33 行中变量 m 存储颜色数值最大区间的累加结果。

小结

（1）全局变量是定义在所有函数元外的变量，能够被所有函数所共享；局部变量是定义在某函数或复合语句内部的变量，只在函数或复合语句内部有效。

（2）静态局部变量有全局变量和一般局部变量双重性质：定义在全局数据区，函数被多次调用，变量的值只被初始化一次，且每次都是在上一次计算结果的基础上执行新的操作；只能在定义的函数体内有效。

（3）全局变量、静态变量、字符串常量存储在全局数据区，函数和代码存储在代码区，函数参数、局部变量、返回地址存放在栈区，动态内存分配在堆区。

（4）如果存在同一工程文件多个文件共同使用的全局变量，就在其中的一个文件中定义全局变量或函数，在其他文件中使用“extern”关键字进行声明，extern 说明的变量或函数为外部文件。

（5）C++程序的作用域按范围由小到大可分为函数原型作用域、局部作用域（块作用域）、函数作用域和文件作用域。除标号外的标识符的作用域都开始于标识符的声明处。

（6）以＃开头、以换行符结尾的行称为预处理命令。预处理命令在程序编译前由预处理器执行。

习题 5

1. 找出下列程序的错误（注：每道题中的源文件都属于相同的工程文件。）

（1）工程文件中包含 func1.cpp 和 func2.cpp 两个源文件。

```
// func1.cpp
   int a = 6;
   int b = 7;
   extern double c;

// func2.cpp
   int a;
   extern double b;
   extern int c;
```

(2) 工程文件中包含 file1.cpp、file2.cpp 和 file3.cpp 三个源文件。

```
// file1.cpp
   int a = 1;
   double func()
   {
        …
   }

// file2.cpp
   extern int a;
   double func();
   void add()
   {
       a = int(func());
   }

// file3.cpp
   extern int a = 2;
   int add();
   main()
   {
       a = add();
   }
```

2. 选择题

(1) 以下叙述正确的是(　　)。

A. 在相同的函数中不能定义相同的名字变量

B. 函数中的形参是静态局部变量

C. 在一个函数体内定义的变量只在本函数范围内有效

D. 在一个函数内的复合语句中定义的变量在本函数范围内有效

(2) 以下叙述不正确的是(　　)。

A. 预处理命令必须以＃开头

B. 凡是以＃开头的语句都是预处理命令行

C. 在程序执行前执行预处理命令

D. ＃define PI＝3.14 是一条正确的预处理命令

(3) 下列程序的运行结果为(　　)。

```
main()
{
  int i = 100;
  {
     i = 1000;
     for(int i = 0;i < 1;i++)
     {
       int i = - 1;
     }
     cout << i;
  }
```

```
  cout <<","<< i;
}
```

A. −1,−1　　B. 1,1000　　C. 1000,100　　D. 死循环

(4) 下列程序的运行结果为(　　)。

```
#indude < iostream >
using namespace std;
int i = 100;
int fun()
{
  static int i = 10;
  return ++i;
}
main()
{
  fun();
  cout << fun()<<","<< i;
}
```

A. 10,100　　B. 12,100　　C. 12,12　　D. 11,100

(5) 下列程序的运行结果为(　　)。

```
#include < iostream >
using namespace std;
void func()
{
  static int i = 1;
  cout <<"i = "<<++i << endl;
}
main()
{
  for(int x = 0;x < 3;x++)
    func();
}
```

A. i=1　　B. i=2　　C. i=1　　D. i=2

i=1　　i=2　　i=2　　i=3

i=1　　i=2　　i=3　　i=4

3. 读程序写结果

```
int a = 1;
int fun()
{ static int a = 1;
  return ++a;
}
main()
{
    fun();
    cout << fun()<<","<< a;
}
```